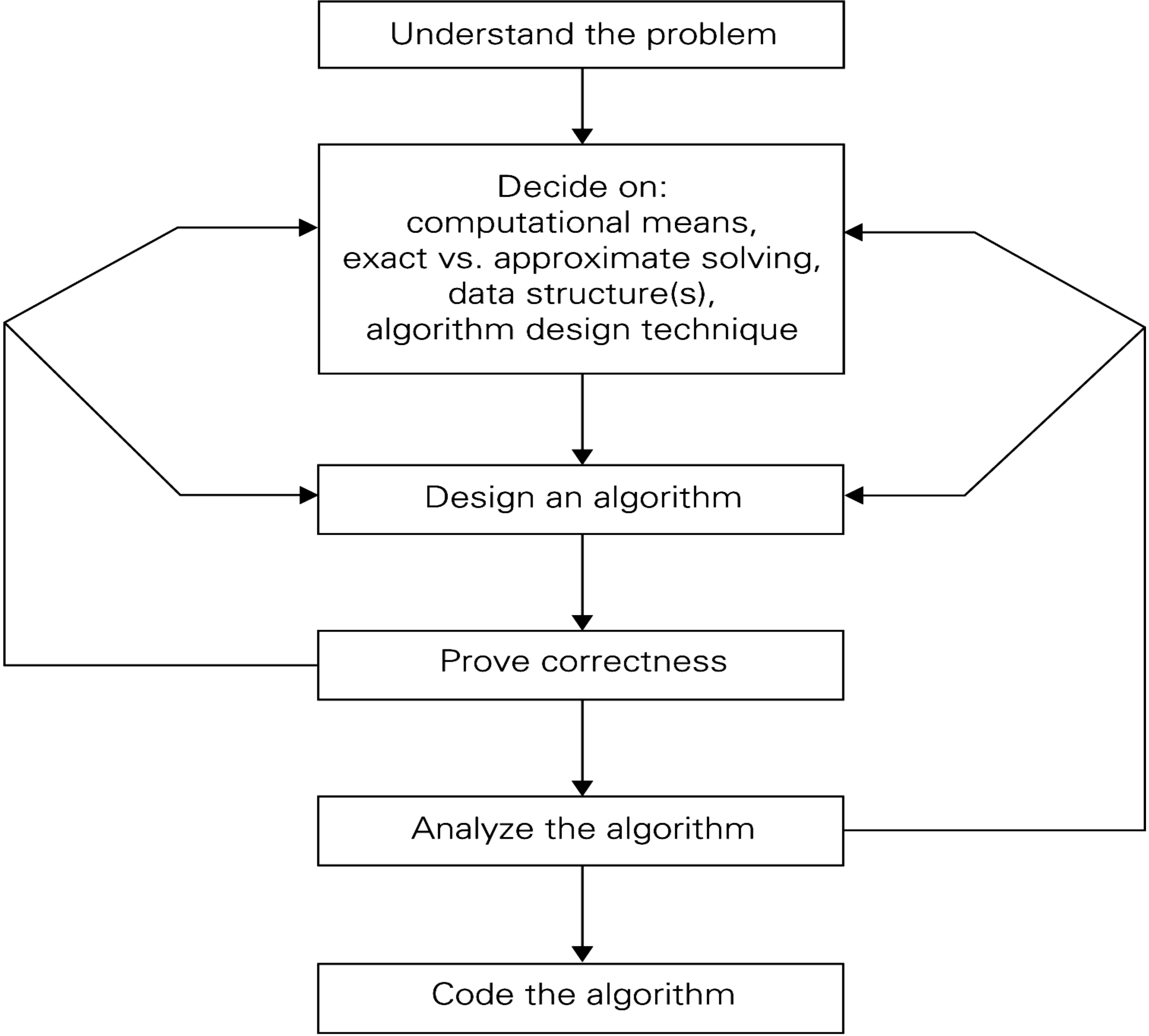
JICSCI803
Algorithms and Data Structures
2019

Highlights of Lecture 08

Numerical Computing
Decidability and Halting Problem

Algorithm Design and Analysis Process



Algorithm Analysis Framework

Measuring an input's size
Measuring running time
Orders of growth (of the algorithm's efficiency function)
Worst-base, best-case and average-case efficiency

Numerical Computing

– Not all computing involves integers (and characters).
– In many applications float (and double) are needed:
  - Science;
  - Engineering;
  - Mathematics.

– **Computation with floating-point numbers requires some special consideration**.

Are computers very powerful?

Can computers compute any numbers?

What is the decimal representation?

decimal representation

A **decimal representation** of a non-negative real number r is an expression in the form of a series, traditionally written as a sum

$$r = \sum_{i=0}^{\infty} \frac{a_i}{10^i}$$

where a_0 is a nonnegative integer, and a_1, a_2, ... are integers satisfying $0 \le a_i \le 9$, called the digits of the decimal representation.

significand

Significand:
the part of a floating-point number that after floating-point;
number in scientific notation that contains its significant digits.

mantissa

1) **The decimal part of a decimal representation.**
   In 2.95424, the mantissa is 0.95424.

2) **The significand; a) ==** mantissa; b) number in scientific notation that contains its significant digits.
in $1.234567^{2} \times 10$， significand = 1.234567

Floating point representation

A floating point number is an approximation to a real number with a finite accuracy. It consists of three parts

The significand (or mantissa) s;

The base b;

The exponent e;

We evaluate the number by calculating $s \times b^e$

In practice, b is fixed so we only need to store s and e.

Floating point representation – Examples

– Let us consider base 10 floating point numbers.

- The real number 123.4567 can be written as $1.234567^{2} \times 10$

  s= 1.234567

  e= 2

- The real number 0. 01234 can be written as $1.234^{-2} \times 10$

  s= 1.234

  e=-2

–We can write floating point numbers using 2 as a base as well.

Change base 10 into base 2 – Examples

How to convert 49 in base 10 into a number in base 2?

49/2=24 remains 1
24/2=12 remains 0
12/2=6 remains 0
6/2=3 remains 0
3/2=1 remains 1
1/2 gets 0 remains 1

Change base 10 into base 2 – Examples

How to convert 0.625 in base 10 into a number in base 2?

0.625
0.625*2=1.25 get 1
0.25*2=0.5 get 0
0.5*2=1 get 1

So 0.625=(0.101)B

Floating point representation – Base 2

– Base 2 floating point works on the same principle – the only difference is that we use binary numbers.

E.g. the real number 17 can be written in binary as

10001.0_2

(Note this is a binary point not a decimal point)

- We can write this in floating point notation as:

$1.0001_2 \times 10_2^e$

s= 1.0001_2

e= 100_2

Floating point representation – Base 2

– Working backwards to base 10

$S = 1.0001_2$

$= 1 + 1/16$

$= 1.0125$

$e = 100_2$

$= 4$

$b = 10_2$

$= 2$

–So our number is $1.0125 \times 2^4 = 1.0125 \times 16 = 17.0$

Floating point representation – Base 2

– If a number has a fractional part things get a little more complicated.

- Consider the decimal number 2.75:
- We can convert the integer part to binary easily $2_{10} = 10_2$
- But what about the fraction?

$.75 = \frac{3}{4} = \frac{1}{2} + \frac{1}{4}$

In binary this is

$.1 + .01 = .11$

- Combining the real and fractional parts we get:

$2.75_{10} = 10.11_2$

Floating point representation – Problems

– Not all real numbers can be expressed in decimal floating point:
  - Transcendental numbers such as π = 3.14159… which never runs out of digits;
  - Fractions such as 1/3 = 0.33' where the 3 repeats forever.

–The same is true for binary floating point.
  - Consider the decimal number 0.1(= 1/10)
  - In binary this is 0.0(0011)' which also repeats.

Floating point representation – Problems

Consider the decimal number 0.1(= 1/10) => ?

In binary this is 0.0(0011)’ which also repeats.

0.1 x 2 = 0.2	integer portion 0
0.2 x 2 = 0.4	integer portion 0
0.4 x 2 = 0.8	integer portion 0
0.8 x 2 = 1.6	integer portion 1
0.6 x 2 = 1.2	integer portion 1
0.2 x 2 = 0.4	integer portion 0
0.4 x 2 = 0.8	integer portion 0
0.8 x 2 = 1.6	integer portion 1
0.6 x 2 = 1.2	integer portion 1

Floating point representation – On a computer

– With a computer we have a limited number of digits (bits) in which we can store a number.

–This effects the accuracy with which we can represent a real number.

–We need to break our word into pieces to store the *significand and* the *exponent*.

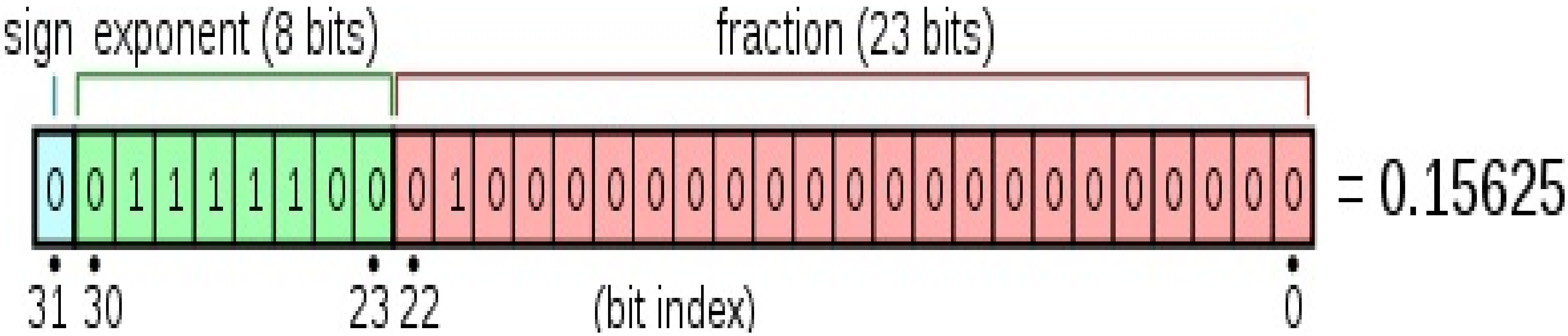
–Exactly how this is done varies from computer to computer and from word size to word size.

Floating point representation – On a computer

The following table shows some standard (IEEE) representations:

Word Length	Sign	Exponent	Significand	Bias*
Half (16 bit)	1	5	10	15
Single (32 bit)	1	8	23	127
Double (64 bit)	1	11	52	1023
Quad (128 bit)	1	15	112	16383

– Note: there is no sign bit for the exponent.
– The bias is explained on the next slide



The real value assumed by a given 32-bit *binary32* data with a given biased *sign*, exponent *e* (the 8-bit unsigned integer), and a *23-bit fraction* is

$$(-1)^{b_{31}} \times (1.b_{22}b_{21}\dots b_0)_2 \times 2^{(b_{30}b_{29}\dots b_{23})_2-127},$$

which in decimal yields

$$\text{value} = (-1)^{\text{sign}} \times \left(1 + \sum_{i=1}^{23} b_{23-i}2^{-i}\right) \times 2^{(e-127)}.$$

In this example:

- $\text{sign} = b_{31} = 0,$
- $(-1)^{\text{sign}} = (-1)^0 = +1 \in \{-1, +1\},$
- $e = b_{30}b_{29}\dots b_{23} = \sum_{i=0}^{7} b_{23+i}2^{+i} = 124 \in \{1, \dots, (2^8-1)-1\} = \{1, \dots, 254\},$
- $2^{(e-127)} = 2^{124-127} = 2^{-3} \in \{2^{-126}, \dots, 2^{127}\},$
- $1.b_{22}b_{21}\dots b_0 = 1 + \sum_{i=1}^{23} b_{23-i}2^{-i} = 1 + 1 \cdot 2^{-2} = 1.25 \in \{1, 1+2^{-23}, \dots, 2-2^{-23}\} \subset [1; 2-2^{-23}] \subset [1; 2).$

thus:

- $\text{value} = (+1) \times 1.25 \times 2^{-3} = +0.15625.$

$$(-1)^{b_{31}} \times (1.b_{22}b_{21}\dots b_0)_2 \times 2^{(b_{30}b_{29}\dots b_{23})_2-127},$$

which in decimal yields

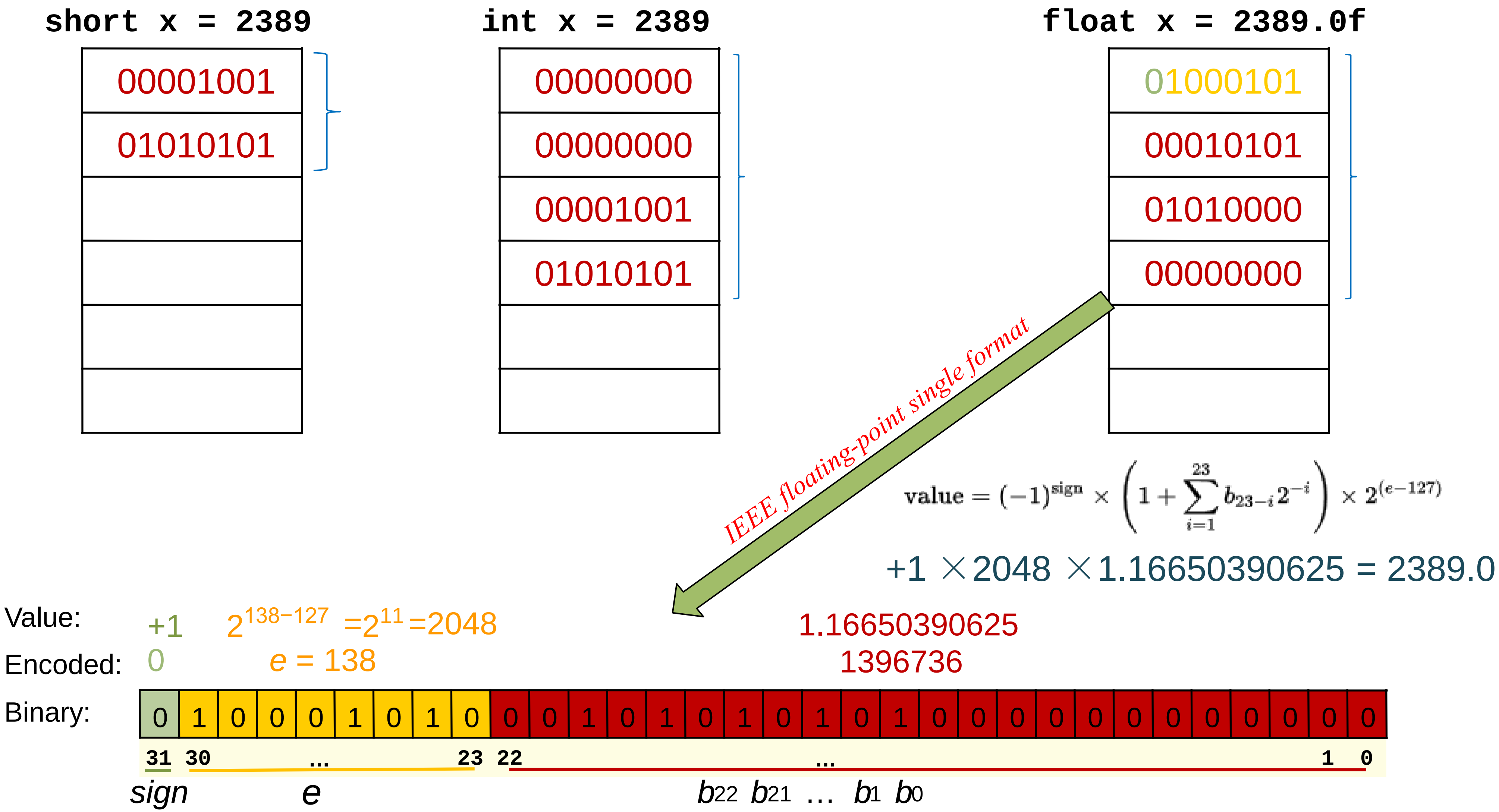
$$\text{value} = (-1)^{\text{sign}} \times \left(1 + \sum_{i=1}^{23} b_{23-i}2^{-i}\right) \times 2^{(e-127)}.$$

In this example:

- $\text{sign} = b_{31} = 0$,
- $(-1)^{\text{sign}} = (-1)^0 = +1 \in \{-1, +1\}$,
- $e = b_{30}b_{29}\dots b_{23} = \sum_{i=0}^{7} b_{23+i}2^{+i} = 124 \in \{1, \dots, (2^8-1)-1\} = \{1, \dots, 254\}$,
- $2^{(e-127)} = 2^{124-127} = 2^{-3} \in \{2^{-126}, \dots, 2^{127}\}$,
- $1.b_{22}b_{21}\dots b_0 = 1 + \sum_{i=1}^{23} b_{23-i}2^{-i} = 1 + 1 \cdot 2^{-2} = 1.25 \in \{1, 1+2^{-23}, \dots, 2-2^{-23}\} \subset [1; 2-2^{-23}] \subset [1; 2)$.

thus:

- $\text{value} = (+1) \times 1.25 \times 2^{-3} = +0.15625$.



short x = 2389
00001001
01010101
int x = 2389
00000000
00000000
00001001
01010101
float x = 2389.0f
01000101
00010101
01010000
00000000
IEEE floating-point single format
$\text{value} = (-1)^{\text{sign}} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right) \times 2^{(e-127)}$
+1 × 2048 × 1.16650390625 = 2389.0
Value:
+1
$2^{138-127} = 2^{11} = 2048$
1.16650390625
Encoded:
0
e = 138
1396736
Binary:
0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
31 30 … 23 22 … 1 0
sign
e
b_{22} b_{21} … b_1 b_0

Floating point representation – On a computer

- We can represent negative and positive exponents without an exponent sign bit by introducing a bias.
  - With this we take the value of the exponent bits and subtract the bias.
  - Exponents of all zeros and all ones are reserved for special purposes.
- Because the significand always has a most significant digit of 1 we do not need to store this bit.
  - This gives us 1 more bit of precision in our floating point numbers.

Computation with floating point numbers

- Because floating point numbers are only approximations to real numbers we can experience a number of problems:
  - overflow;
  - underflow;
  - rounding.

Overflow errors

– The finite size of the **exponent** part of a floating point number means that we can only represent numbers with a **maximum size** related to this.

– Using 16 bit floating point numbers as an example:
  - Internal representation 0111101111111111
  - Sign bit is 0 so the number is positive
  - Exponent is 11110 = 30 –(bias of 15) = 15
  - Significand is 1.1111111111 (leading bit is implied)
  - So the number is $(1+1023/1024)^{15} \times 2 = 65504.0_{10}$
  - If we multiply this number by 2 the exponent is too big to store – we have an overflow.

Underflow errors

– Similarly, we can only represent numbers with a ***minimum size*** related to the size of the **exponent** field.

– Using 16 bit floating point numbers as an example:

- Internal representation 0000010000000000
  - Sign bit is 0 so the number is positive
  - Exponent is 00001 = 1 –(bias of 15) = –14
  - Significand is 1.0000000000 (leading bit is implied)
  - So the number is (1) $\times 2^{-14}$ = 0.00006103515625_{10}
  - If we divide this number by 2 the exponent is too small to store –we have an underflow.

Rounding errors

– These arise because we have ***a finite number of bits*** in which to store the significand.

- Consider (16 bit)

  0011110000000001 $\times$ 0100001000000000

  = 1025/1024 $\times$ 3

  = 3075/1024

– If we convert this to 16 bit floating point we would need 11 bits to store the significand.

– Because we only have 10 bits the final result is stored as 3076/1024 – we have a rounding error.

Special values

- As noted earlier, floating point numbers with all exponent bits equal to zero and with all exponent bits equal to one are reserved for special cases.
- All-zero exponent floating point numbers are used to represent so-called subnormal numbers which are used to reduce the incidence of underflow errors.
- All-one exponent floating point numbers are used to represent things like positive and negative infinity and Not-a-Number(the value that results from certain operations with undefined results).

Questions

What can we do if we want to deal with “larger” numbers?

Function Evaluation

– What follows involves the evaluation of functions so we will look at this now.

– In particular we will look at the evaluation of polynomial functions.

– These are of the form:

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x_3 \ldots + a_n \, x^n$$

– This polynomial function is said to be of order n.

– We can store the coefficients a0, a1… in an array.

Polynomial Evaluation

– Consider the following code:

```
function eval1 (x, a[0..n])
   value = 0
   power = 1
   for i = 0 to n
      value = value + a[i] * power
      power = power * x
   end for
   return value
end eval1
```

Polynomial Evaluation

- To evaluate a function of order *n* we perform *n* additions and 2*n* multiplications
  - Is this the best we can do?
- Let us rewrite the function as follows:

  $f(x) = a0 + x(a1+x(a2+x(a3\ldots x(an)\ldots)))$
- If we now code the function evaluation using this scheme we get the following code:

Polynomial Evaluation

– Consider the following code:

```
function eval2 (x, a[0..n])
     value = a[n]
     for i = n-1 down to 0
       value = a[i] + value * x
     end for
     return value
end eval2
```

–This involves *n* additions and only n multiplications.

Root Finding

- We often need to find a value of *x* for which a function takes the value 0.
- Such *x* values are called the roots of the equation.
- For example the roots of the order 2 equation

$$f\,(x) = x2–5x+ 6$$

are

$$x= 2 \text{ and } x= 3.$$

- (Note: a0= 6, a1= 5, a2= 1)

Root Finding

– For order 2 equations we can find the root directly using the well-known quadratic formula

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

– Where a is a_2, b is a_1 and c is a_0.

– If the order is greater than 2 or the function is not a polynomial, finding a root may be much harder.

Numerical Root Finding

– Assume we have some function *f*(*x*) and we wish to find a value of x for which f(x) = 0.
– We can approximate such a root by using an iterative process.
  - This can be done in a number of ways.
  - We will consider 2 such ways
    – The interval bisection method
    – The ***regula falsi*** method
  - Both techniques have a common starting point.

Numerical Root Finding

– If we have 2 values of x, x_1 and x_2 such that $f(x_1) < 0$ and $f(x_2) > 0$ then it should be obvious that some value of x, x_r between x_1 and x_2 must be a root of the function.

– (Note: x_1 does not have to be less than x_2).

– We can use this as the basis of our root finding algorithm.

– Let us assume that we have a function f_eval(x) already defined which returns the value of f(x).

The interval bisection method

– Given x_1 and x_2 as already defined we calculate the value of *x* half way between them x_{mid}.

$$x_{mid} = 0.5(x_1 + x_2)$$

– If we evaluate $f(x_{mid})$, three possibilities exist:

i. $f(x_{mid}) = 0$ and we have found our root

ii. $f(x_{mid}) < 0$ and a root must lie between x_{mid} and x_2

iii. $f(x_{mid})$, > 0 and a root must lie between x_{mid} and x_1

– In cases *ii* and *iii* we can replace one of starting values with the midpoint value and try again.

– Each iteration will bring us closer to the root.

The interval bisection method in code:

```
function b_root(x_1, x_2)
   f_1 = f_eval(x_1)
   f_2 = f_eval(x2)
   repeat
      x_mid = (x_1 + x_2) / 2
      f_mid = f_eval(x_mid)
      if (f_1 * f_mid > 0) then
         x_1 = x_mid
         f_1 = f_mid
      else
         x_2 = x_mid
         f_2 = f_mid
      endif
   until f_mid is close to 0
   return x_mid
end
```

Stopping the process

- In practice, we almost never get a value of $x_{\mathtt{mid}}$ for which $f(x_{\mathtt{mid}})$ is exactly 0.
- This is why the code on the previous slide used the test

```
until f_mid is close to 0
```

  to terminate.
- This is usually a test based on some pre-set tolerance which will depend on how close to the correct answer we need to get.
- The actual code is usually something like

```
until abs(f_mid) < tolerance
```

The method of *regula falsi*

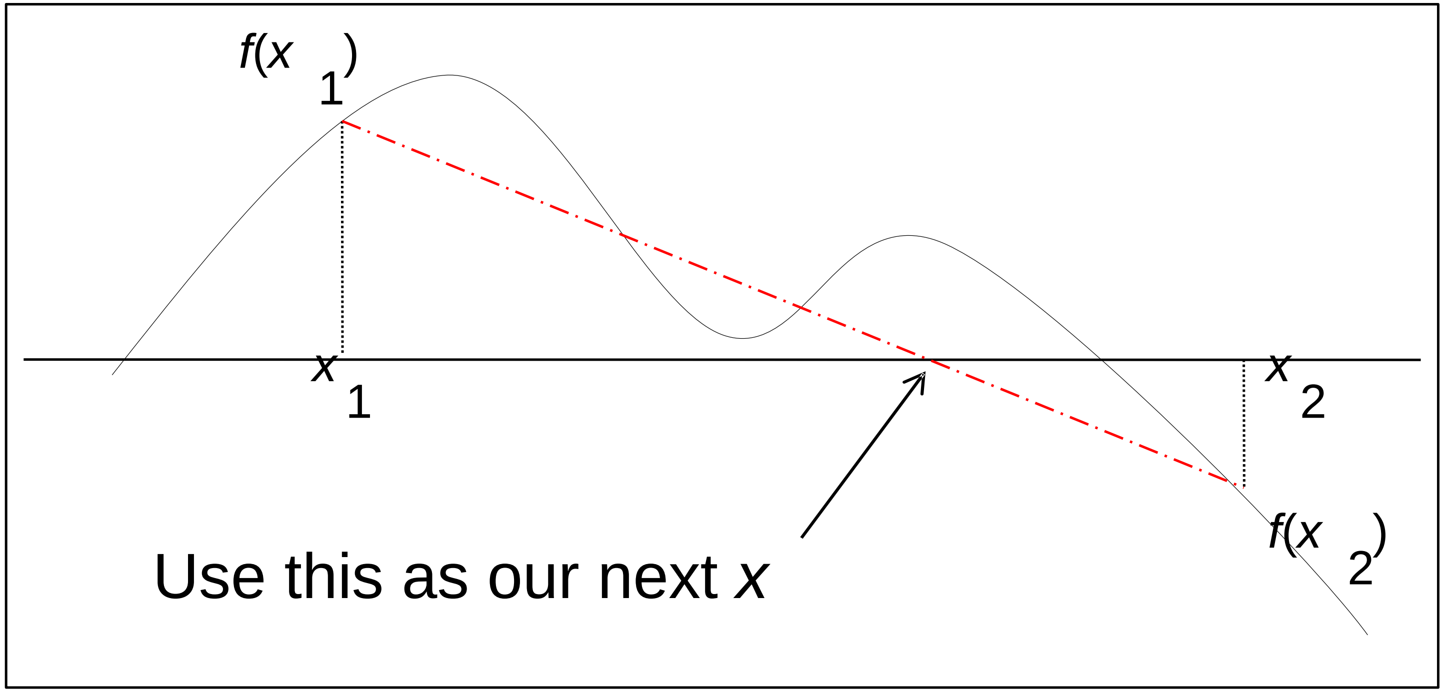
– The choice of the mid point between x_1 and x_2 as the next x value is an arbitrary one.

– Any value between x_1 and x_2 could be used.

– Is there a better choice than $x_{\mathbf{mid}}$?

–Let us consider the question with a picture–perhaps this will give us a clue.

Consider the following graph of $f(x)$



Regula *falsi*

– We can use similar triangles to determine the correct value for x_{new}

$$\frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{f(x_2) - 0}{x_2 - x_{new}}$$

which gives

$$x_{new} = x_2 - \frac{f(x_2)}{f(x_2) - f(x_1)} \times (x_2 - x_1)$$

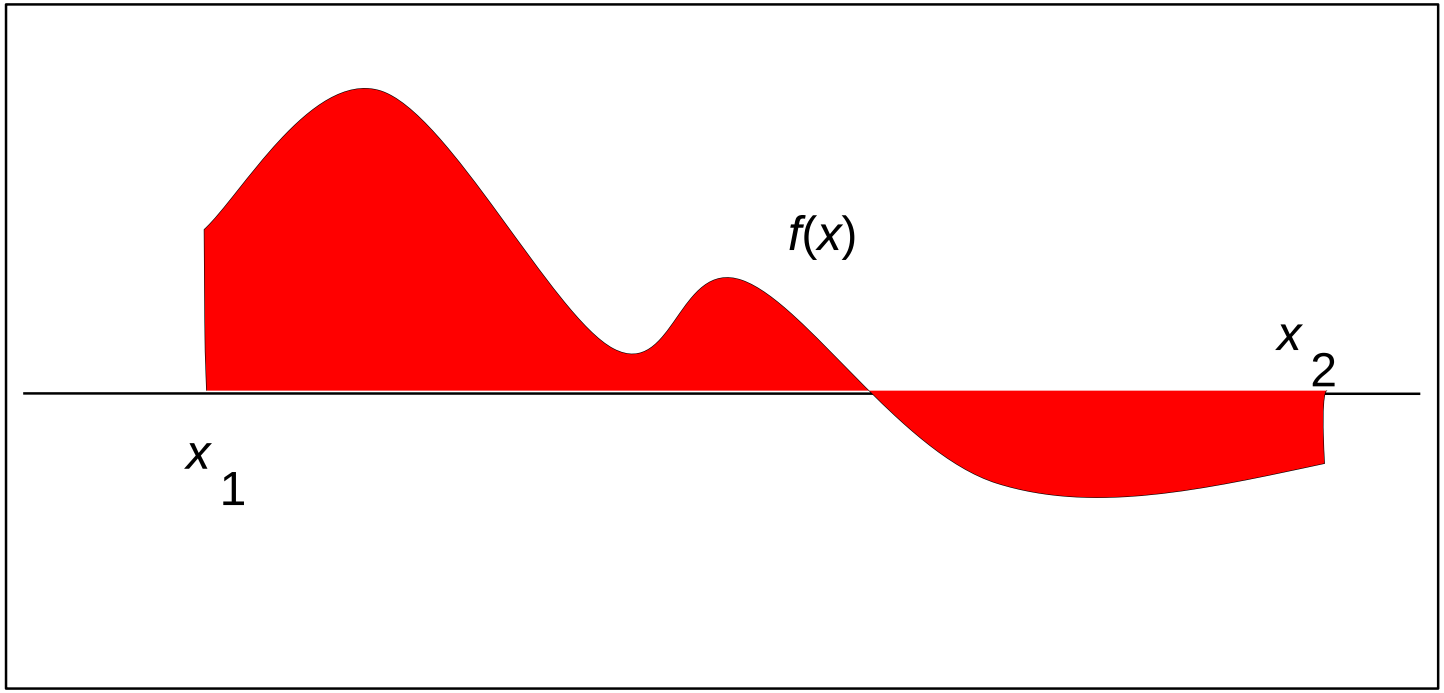
– We can then proceed exactly as in the bisection method using x_{new} instead of x_{mid}

Area under a curve (numerical integration)

– Given a function $f(x)$, we want to find the area between the curve and the x-ax is over some range $x_1 < x < x_2$

– We can represent this with a picture

The area we wish to find is shaded in red



We can get an approximate answer numerically in a number of ways

– Among these are:

i. Estimate the area with a series of rectangular segments

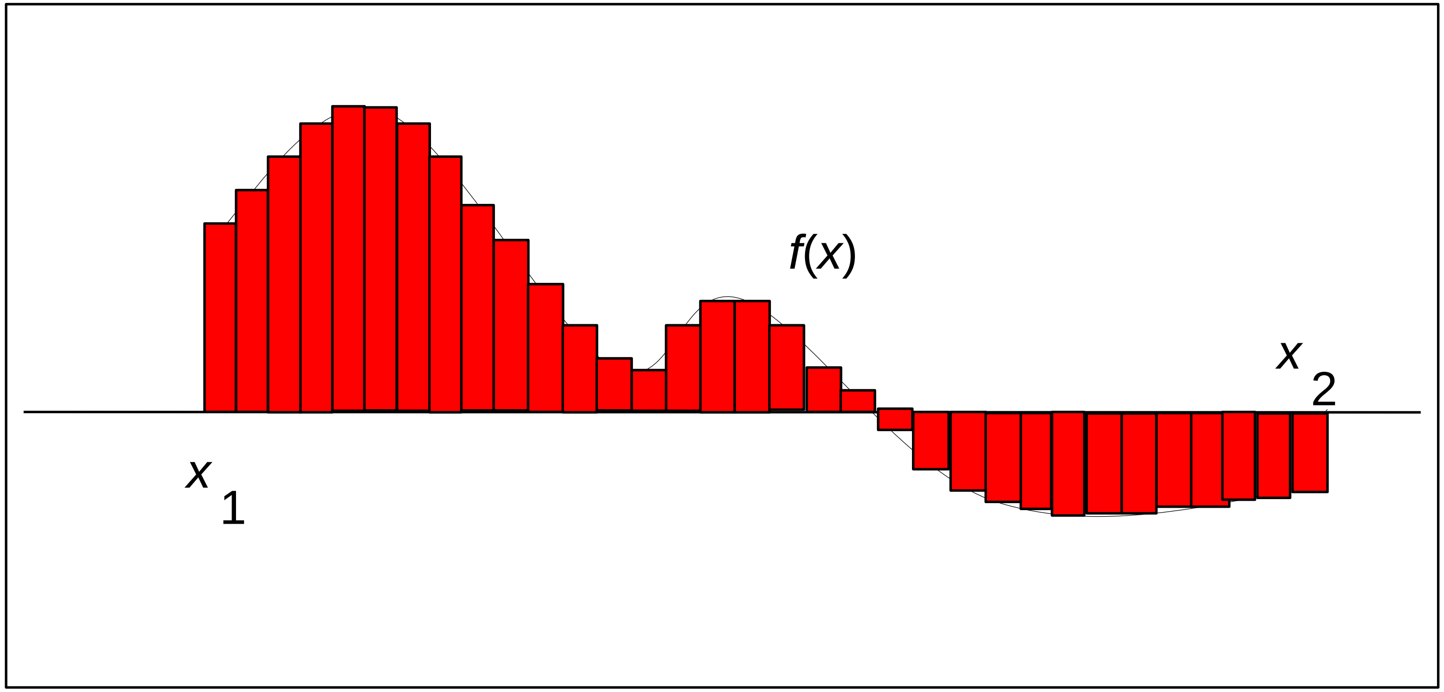
ii. Estimate the area with a series of trapezia.

– We will look at these methods in turn.

Approximation using rectangles

- We divide the interval from $x1$ to x2 into n segments, each of width w where w = (x2–x1)/n
- For each interval we construct a rectangular box with width w and height h.
- The height of each rectangle is the value of $f(x)$ at the mid point of the box.
- The sum of the areas of these boxes is approximately equal to the area we are looking for.

- The approximate area is shaded in red.



- The more strips we use, the more accurate our estimate becomes.

```
function area_by_rectangles(x1, x2)
    n = 1
    xmid = 0.5*(x1+x2)
    a = (x2-x1) * abs(f_eval(xmid))
    repeat
      area = a
      n = 2*n
      w = (x2-x1)/n
      xstripmid = x1 -w/2
      a = 0.0
      for i = 1 to n
         xstripmid = xstripmid + w
         a = a + w * abs(f_eval(xstripmid))
      end for
   until a is close to area
   return a
end
```

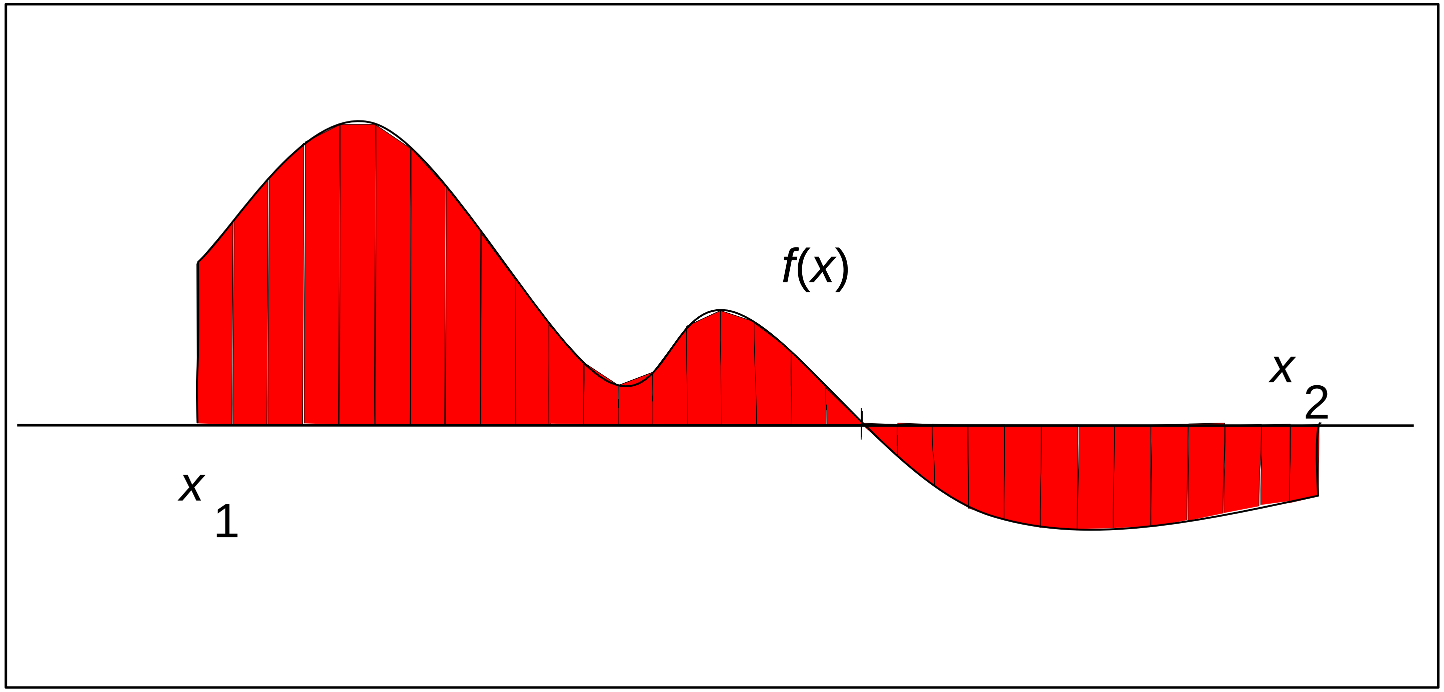
Approximation using trapezia

– We again divide the interval from x_1 to x_2 into n segments, each of width w= $(x_2–x_1)/n$.

–This time we construct trapezia which fit between the x-ax is and the curve at each end of the interval.

–The sum of the areas of these trapezia is approximately equal to the area we are looking for.

- The approximate area is shaded in red.



- You can see that this yields a more accurate estimate for a given value of *n*.

```
function area_by_trapezia(x1, x2)
   n = 1
   a = (x2-x1) * 0.5 * abs(f_eval(x1) + f_eval(x2))
   repeat
      area = a
      n = 2*n
      w = (x2-x1)/n
      xstart = x1
      fstart = f_eval(xstart)
      a = 0.0
      for i = 1 to n
         xend = xstart + w
         fend = f_eval(xend)
         a = a + w * 0.5 * abs(fstart + fend)
         xstart = xend
         fstart = fend
      end for
   until a is close to area
   return a
end
```

Decidability and the Halting Problem

Outline

What can be computed?

Can we solve all problems?

Are there problems we can't solve?

We will look at the more famous *halting problem*

Decidable Problems

We have now defined the class of all computable algorithms

all of which can be programmed in C++ and run on any modern processor

Are there problems which cannot be solved with algorithms?

Decidable Problems

To begin, we will define a *decision problem* to be a question with a yes-or-no answer

- Turing's 1937 paper referred to *Entscheidungs problem*, or *decision problems*
- Specifically, he showed that there exist decision problems which cannot be computed
- We will look at the *halting problem*

Undecidable Problems

The halting problem:

Given a function `f`, is it possible to write a Boolean-valued function

```
bool does_halt( f, x );
```

which returns true if `f(x)` does not go into an infinite loop?

It sounds plausible...

Even Microsoft has a research group looking into this problem: http://research.microsoft.com/TERMINATOR/

Undecidable Problems

We will show that it is impossible to write such a function

We will assume such a function exists and then show that this leads to a logical contradiction

Undecidable Problems

Suppose that `does_halt` exists, in which case, we may define a second function that calls `does_halt`:

```
paradox := proc( f )
    if does_halt( f, f ) then
        # If f(f) is said to finish execution,
        #   paradox goes into an infinite loop
        from 1 to infinity do end do
    else
        # If f(f) is said to go into an infinite loop,
        #   paradox return immediately
        return
    end if
end proc:
```

Undecidable Problems

To summarize, our function `paradox( f )` is one that:

Returns if `f(f)` is said to go into an infinite loop,

Otherwise, `paradox` itself goes into an infinite loop

What should be the return value of

`does_halt( paradox, paradox)`?

Does `paradox( paradox )` go into an infinite loop?

Undecidable Problems

Assume that `does_halt( paradox, paradox)` returns **`true`**

`does_halt` determined that `paradox( paradox )` finishes

In this case, `paradox( paradox )` goes into an infinite loop

Therefore `does_halt( paradox, paradox )` cannot return `true`

```
paradox := proc( f )
if does_halt( f, f ) then
        from 1 to infinity do end do
    else
        return
    end if
end proc:
```

Undecidable Problems

Alternatively, assume that `does_halt( paradox, paradox)` returns **`false`**

- `does_halt` determined that `paradox( paradox )` loops infinitely often

In this case, `paradox(paradox)` returns immediately

- Therefore `does_halt( paradox, paradox )` cannot return `false`

```
paradox := proc( f )
    if does_halt( f, f ) then
        from 1 to infinity do end do
    else
        return
    end if
end proc:
```

Undecidable Problems

Thus, we have a logical contradiction:

Whatever such a function `does_halt( paradox, paradox )` returns, it will be incorrect

- If it returns true, `paradox(paradox)` goes into an infinite loop
- If it returns false, `paradox(paradox)` halts

We use *reductio ad absurdum*: $(x \rightarrow \neg x) \rightarrow (\neg x)$

Therefore, a function like `does_halt` cannot exist

Therefore, it is not possible to find a computational answer to the halting problem

Summary

Regarding decidability

Not everything is decidable

Some problems cannot be solved with algorithms

It may be possible to guarantee that problems are solvable if we restrict the possible range of inputs

References

Wikipedia,
http://en.wikipedia.org/wiki/Decidability_(logic)
Wikipedia, http://en.wikipedia.org/wiki/Halting_problem

Discussions

1. How computers store characters and numbers ?

Homework

Read materials on
Godel incompleteness theorem